Implementing Java-like languages in Xtext with Xsemantics

Lorenzo Bettini

Dipartimento di Informatica, Università di Torino, Italy

PISA, CINA Kick-off Meeting, 5 Feb 2013.
**Xtext**  Eclipse framework

- **Xtext** provides a higher-level framework that generates most of the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse.
  - Really quick and easy to have a working implementation
  - Implement while designing and formalizing
**Xtext** Eclipse framework

- **Xtext** provides a higher-level framework that generates most of the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse.
  - Really quick and easy to have a working implementation
  - Implement while designing and formalizing

- Type system and reduction rules still implemented in Java
- Gap between the formalization and implementation
Featherweight Java

- a lightweight functional version of Java:
  - mutually recursive class definitions,
  - class inheritance,
  - object creation,
  - method invocation,
  - method recursion through this,
  - subtyping and
  - field access.

A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems, 23(3):396–450, 2001.
**Xtext** Eclipse framework

- Write the grammar of the language using an EBNF-like syntax
- **Xtext** generates an ANTLR parser.
- During parsing, the AST is generated in the shape of an EMF model
**Xtext** Eclipse framework

- Write the grammar of the language using an EBNF-like syntax
- **Xtext** generates an ANTLR parser.
- During parsing, the AST is generated in the shape of an EMF model

**Selection**: receiver=Expression '.' message=[Member]

```
  ( '(' (args+=Expression (',' args+=Expression)*)? ')' )? ;
```

Write the grammar of the language using an EBNF-like syntax.

**Xtext** generates an ANTLR parser.

During parsing, the AST is generated in the shape of an EMF model.

**Selection:** `receiver=Expression '.' message=[Member]`  
`( '(' (args+=Expression (',', args+=Expression)*))? ' )' )` ;

```java
interface Selection extends Expression {
    Expression getReceiver();
    Member getMessage();
    EList<Expression> getArgs();
}
```
Featherweight Java grammar in Xtext

Program: (classes += Class)* (main = Expression)?;
Class: 'class' name=ID ('extends' superclass=[Class])? '{'
  (members += Member)* '}'
Member: Field | Method;
Field: type=[Class] name=ID ';
Method: type=[Class] name=ID
  '(' (params+=Parameter (',', params+=Parameter)*)? ')
  '{' body=MethodBody '}
Parameter: type=[Class] name=ID;
TypedElement: Member | Parameter;
MethodBody: 'return' expression=Expression ';

Expression: Selection | TerminalExpression ;
Selection: receiver=Expression '.' message=[Member]
  ('(' (args+=Expression (',', args+=Expression))*? ')')? ;
TerminalExpression: This | ParamRef | New | Cast | Paren ;
This: variable='this';
ParamRef: parameter=[Parameter];
New: 'new' type=ClassType
  ('' (args+=Expression (',', args+=Expression))*? '')';
Cast: '(' type=[Class] ')' expression=TerminalExpression;
Paren: '(() Expression ')';
Figure: A screenshot of the FJ IDE.
Checking that a program is semantically correct in the XTEXT:

- **scoping**: cross references can be resolved, e.g., the binding of a variable to its definition in the program

```java
class A {
    String s;
    String toString() {
        return this.s;
    }
}
```
Checking that a program is semantically correct in the XTEXT:

- **scoping**: cross references can be resolved, e.g., the binding of a variable to its definition in the program

```java
class A {
    String s;
    String toString() {
        return this.s;
    }
}
```

- **validation**, the AST model is correct, e.g., checking that the return value of a method body is consistent with the method signature

```java
class A {
    String s;
    String toString() {
        return 10;
    }
}
```
Our aim

- Scoping and Validation usually rely on types;
- the programmer needs to implement a type system in Java.
- Instead, we would like to implement the type system functionalities with a DSL.
- This would allow us to have the typing rules in a compact form,
- and then have the corresponding Java code

Our proposal

We present XSEMANTICS, a DSL for writing rules for languages implemented in Xtext

- the static semantics (type system),
- the dynamic semantics (operational semantics) and
- relation rules (subtyping).
A system definition in **Xsemantics** is

- a set of *judgments*
- and a set of *rules* which have a conclusion and a set of premises (and a rule environment)
A system definition in **XSEMANTICS** is

- a set of *judgments*
- and a set of *rules* which have a conclusion and a set of premises (and a rule environment)

Starting from the definitions of judgments and rules, **XSEMANTICS** generates Java code that can be used for scoping and validation.
XSEMANTICS and Xbase

- an extensible and reusable statically typed expression language
- a Java with “less noise”
  - type inference
  - closures
- integrates completely with Java and Eclipse JDT
- full access to Java type system
Xsemantics and Xbase

- an extensible and reusable statically typed expression language
- a Java with “less noise”
  - type inference
  - closures
- integrates completely with Java and Eclipse JDT
- full access to Java type system

Example

```java
val personList = new ArrayList(
    new Person("James", "Smith", 50),
    new Person("John", "Smith", 40),
    new Person("James", "Anderson", 40),
    new Person("John", "Anderson", 30),
    new Person("Paul", "Anderson", 30))

personList.filter(firstname.startsWith("J"))
    .sortBy(age).take(3).map(surname + ", " + firstname).
    join("; ")
```
FJ judgments in XSEMANTICS

judgments {
  type |- Expression expression : output Class
    error "cannot type " + expression
  subtype |- Class left <: Class right
    error left + " is not a subtype of " + right
  subtypesequence |- List<Expression> expressions << List<? extends TypedElement> elements reduce |- Expression exp ~> output Expression
}
rule MyRule
    G |- Selection exp : Class type
from 
    // premises
    type = ... // assignment to output parameter
**XSEMANTICS rules**

`rule` MyRule
   - Selection exp : Class type
`from` {
   // premises
   type = ... // assignment to output parameter
}

`rule` MyRule
   - Selection exp : exp.message.type
`from` {
   // premises
}

must "respect" the judgment judgments
   - Expression expression : output Class error "cannot type " + expression ...
**XSEMANTICS rules**

```plaintext
rule MyRule
    G |- Selection exp : Class type
from {  
    // premises
    type = ... // assignment to output parameter
}
rule MyRule
    G |- Selection exp : exp.message.type
from {  
    // premises
}

must “respect” the judgment

judgments {
    type |- Expression expression : output Class
    error "cannot type " + expression
    ...
```
**XSEMANTICS axioms**

**axiom** TThis

\[ G \vdash \text{This \_this : env}(G, \ 'this' , \text{Class}) \]
XSEMANATICS axioms

**axiom TThis**

\[ \text{TThis} \quad G |- \text{This } \_\text{this} : \text{env}(G, 'this', \text{Class}) \]

must “respect” the judgment

**judgments** {

\[ \text{type} |- \text{Expression expression : output Class} \]

\[ \text{error "cannot type " + expression} \]

...
Rule for subtyping

```plaintext
rule Subclassing
    G |- Class left <: Class right
(from)
    left == right or
    right.name == "Object" or
    G |- left.superclass <: right
```
Examples

Grammar for FJ

Cast: `( type=[Class] )` expression=Expression;
Grammar for FJ

```
Cast: '()' type=[Class] ')' expression=Expression;
```

Rule for cast

```
rule TCast
    G |- Cast cast : cast.type
from { 
    G |- cast.expression : var Class expType

    { G |- cast.type <=: expType }
    or 
    { G |- expType <=: cast.type }
}
```
rule SubtypeSequence
G |- List<Expression> expressions << List<TypedElement> elems
from {
    expressions.size == elems.size
    var i = 0
    for (exp : expressions) {
        G |- exp : var Class expType
        G |- expType <= elems.get(i++)
    }
}
Examples

**Rule for “new”**

```plaintext
rule TNew
  G |- New newExp : newExp.type
from {
  var f = fields(newExp.type)
  G |- newExp.args << f
}
```

```plaintext
rule SubtypeSequence
G |- List<Expression> expressions << List<TypedElement> elems
from {
  expressions.size == elems.size
  var i = 0
  for (exp : expressions) {
    G |- exp : var Class expType
    G |- expType <: elems.get(i++)
  }
}
```
Checkrules

For generating the Validator

```scala
checkrule CheckMethodBody for Method method
from {
  val C = method.getContainerOfType(typeof(Class))
  'this' <- C | method.body.expression : var Class bodyType
  empty |- bodyType <: method.type
}
```
Errors and Traces

- in case one of the premises fails the whole judgment fails
- the error trace will be used to automatically generate all the error markers
- in general the trace of the rules applied is available to the programmer for testing and debugging
Using traces: Example

Expressing *Subject Reduction*

```plaintext
judgments {
  subjred |= Expression e ~> output Expression : output Class <: output Class
}

rule SubjRed
  G |= Expression e ~> Expression e1 : Class C1 <: Class C
from {
  G |- e : C
  G |- e ~> e1
  G |- e1 : C1
  G |- C1 <: C
}
```
Using traces: Example

class A {
    Object m() { return this.n(new B()); }
    A n(A o) { return new A(); }
}

class B extends A {
}

new A().m()
Using traces: Example

class A {
    Object m() { return this.n(new B()); }
    A n(A o) { return new A(); }
}
class B extends A {}

new A().m()
Case study: type inference for λ-calculus

- we also developed a prototype implementation of λ-calculus in Xtext;
- we used XSEMATICS to write a type system for inferring types with type variables (generic types);
- we implemented unification in order to infer the most general type.
Case study: type inference for $\lambda$-calculus

- we also developed a prototype implementation of $\lambda$-calculus in Xtext;
- we used Xsemantics to write a type system for inferring types with type variables (generic types);
- we implemented unification in order to infer the most general type.

\[
\begin{align*}
\text{lambda} & \ \ x . \ x & \ a \rightarrow a \\
\text{lambda} & \ \ x . \ 10 & \ a \rightarrow \text{int} \\
\text{lambda} & \ \ x . \ -x & \ \text{int} \rightarrow \text{int} \\
(\text{lambda} & \ \ x . \ \text{lambda} \ y . \ y) \ 10 \ & \ a \rightarrow a \\
\text{lambda} & \ \ x . \ \text{lambda} \ y . \ x \ y & \ (a \rightarrow b) \rightarrow a \rightarrow b \\
\text{lambda} & \ \ x . \ \text{lambda} \ y . \ y \ x & \ a \rightarrow (a \rightarrow b) \rightarrow b \\
\text{lambda} & \ \ f . \ (\text{lambda} \ x . \ (f \ (f \ x))) \ & \ (a \rightarrow a) \rightarrow a \rightarrow a \\
\text{lambda} & \ \ f . \ \text{lambda} \ g . \ \text{lambda} \ x . \ (f \ (g \ x)) \ & \ (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b
\end{align*}
\]
Inferring $\lambda$ types

we can use the generated type system to write Eclipse editor actions for automatically inserting the inferred types of $\lambda$ terms:
Inferring $\lambda$ types

we can use the generated type system to write Eclipse editor actions for automatically inserting the inferred types of $\lambda$ terms:
Inferring $\lambda$ types

we can use the generated type system to write Eclipse editor pop-up actions for automatically inferring the types of terms:
Fully featured IDE
Debugging rules

```
G |- Expression e --> Expression e1 : Type T1 <=: Type T
  from {
    G |- e : T
    G |- e --> e1
    G |- e1 : T1
    G |- T1 <=: T
  }

// checkrules (for the generated validator)
```
Conclusions

http://xsemantics.sourceforge.net


Conclusions

http://xsemantics.sourceforge.net

Lorenzo Bettini.
A DSL for Writing Type Systems for XtextLanguages.

Lorenzo Bettini.
Implementing Java-like languages in Xtext with Xsemantics.
To appear.

Lorenzo Bettini, Dietmar Stoll, Markus Völter, and Serano Colameo.
Approaches and Tools for Implementing Type Systems in Xtext.

Thanks!